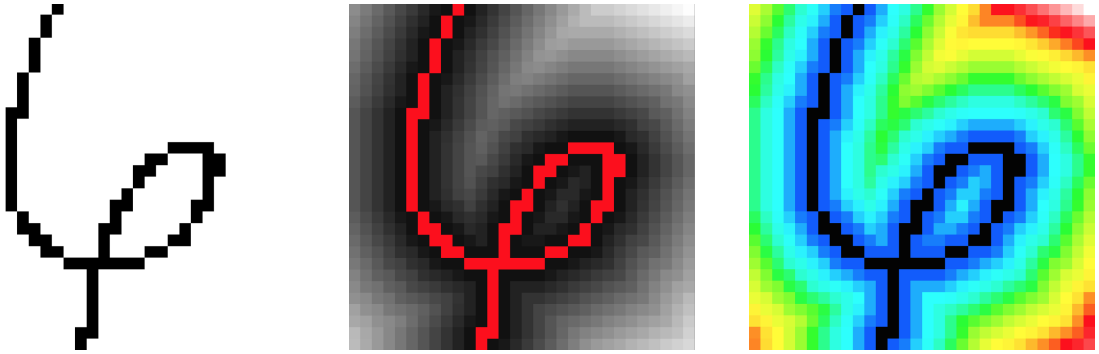
Distance Fields & Priority Queues

For this assignment you will manipulate 2D images and calculate and visualize the *distance field* from a shape drawn in a black and white input image. A distance field is scalar valued function defined for every point in space (in our case, for every pixel in a simple 2D grid). The value of the distance field at a point in space is simply the distance from that point to the closest point on the input shape. Distance fields are useful in many applications including collision detection in 3D simulations, shape modeling and registration/similarity detection, motion planning for robotics, and computer vision and image processing.



In the example above we start with a 30x30 black and white image (left). The black pixels represent the shape from which we want to calculate the distance field. All black pixels have distance field value=0.0. The 4 pixels adjacent to any black pixel (that are not black themselves) have distance field value=1, the 4 diagonal neighbors have distance value=$\sqrt{2}$, pixels 2 units in the horizontal or vertical direction have distance value=2.0, etc. In the middle image we see a visualization of this distance field. Pixels with distance value=0 are colored red=<255,0,0>. The remaining distance values are mapped to shades of grey such that smaller values are darker and the largest distance value is colored white=<255,255,255>. The right image above shows an alternate visualization where the same distance values are mapped to a more colorful scale.

We provide you with a simple `Image` class that loads and saves (uncompressed) .ppm files and the implementation of a naive (but very slow) algorithm for computing the distance field. For this assignment you will analyze the performance of the naive algorithm, and implement a significantly faster algorithm using a priority queue. *Be sure to read the entire handout before beginning your implementation.*

Naive Distance Field Algorithm

The naive algorithm consists of nested loops that compare every pixel to every other pixel and computes the distance between each pair of pixels. Each pixel remembers the shortest distance to a black pixel. The provided code implements this algorithm and can be run with the following command line:

```
distancefield.exe squiggle_30x30.ppm out.ppm naive_method greyscale
```

Your first task is to analyze the code and determine the order notation of the algorithm in terms of w & h, the width and height of the input image, and p, the number of black pixels in the input image. Run a variety of different size test cases using the naive algorithm and tabulate the running times. On UNIX/Linux/MacOSX/Cygwin add the "`time`" function to the front of your command line:

```
time distancefield.exe squiggle_30x30.ppm out.ppm naive_method grey_bands
```

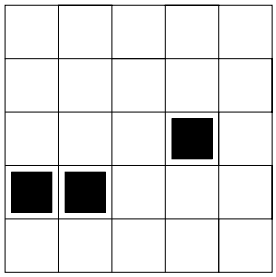
When the program finishes, you'll be given an estimated breakdown of the running time of your program: "real" (the total time), "user" (just your program), and "sys" (time when your program is waiting for resources, e.g., writing to a file, or waiting for the CPU when other programs are running simultaneously). Once you have collected the data, analyze the results. Do they match your predicted order notation? Record your order notation, running time data, and discuss in your `README.txt` file.

Improvement on the Naive Algorithm

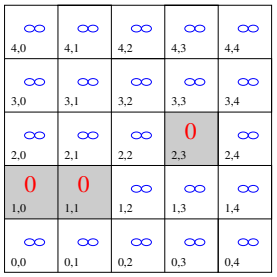
Rather than comparing every pixel to every other pixel, we can compare every pixel to *every black pixel*. The change in implementation is minor, but will likely have a rather large impact on the performance. Go ahead and make the change to the code, predict the order notation of the improved version (again in terms of w, h, and p), and run the same experiments to confirm your predictions.

The Level Sets Fast Marching Method

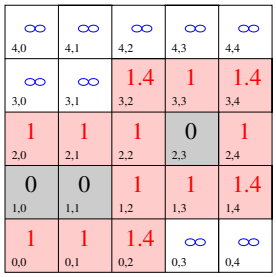
But wait, there's more! The problem of calculating the distance field can be completely reformulated. Instead let's track or march an *advancing front* from the input shape outwards. The method is illustrated in a series of diagrams below for a simple 5x5 pixel input image.



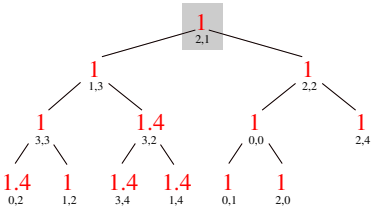
input image



initialization of the signed distance field



propagating initial values

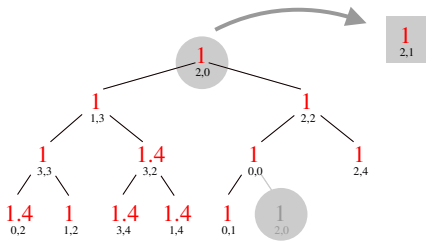


initial priority queue of pixels

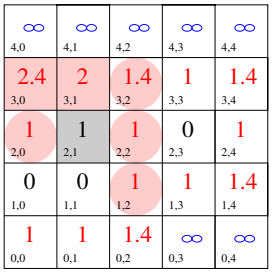
In the first step, we collect all black pixels from the input image and initialize the distance field value for these pixels to 0. The distance value for all other pixels is initialized to infinity (a very large number works fine too). Next, for each black pixel, b, we consider each of the 8 neighbor pixels, n. If the current distance value of n is greater than the distance value of b + the distance between b & n, we update the distance value of n to be the distance value of b + the distance between b & n. We call this *propagating* the distance data from one pixel to its neighbors. Note that the calculated distance is not necessarily the correct final answer for the distance of the pixel n, but it is a conservative upper bound on that value.

The collection of newly-updated pixels (labeled in red in the third diagram above) are placed into a priority queue, using the distance value as the priority value. Smaller distance values will percolate up to the top of the heap. The heap contains only the advancing front of estimated distance value pixels (marked red), not the known values (marked black) or the unknown values (marked blue).

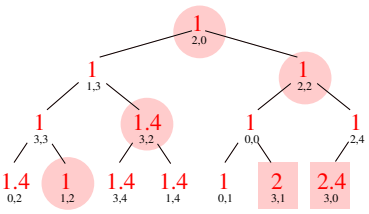
Now the real fun happens. To continue progressing this front of estimated values across the image, we will first grab the top pixel from the priority queue, (2,1). Because no other pixel in the priority queue has smaller distance value, we can guarantee that its distance will not change when information is propagated from its red-marked neighbors. Therefore, we can set this pixel's distance value as known (mark it black), remove it from the priority queue, and propagate its distance information to its neighbors, as shown below:



after popping & fixing the top value, grab the last leaf & percolate down

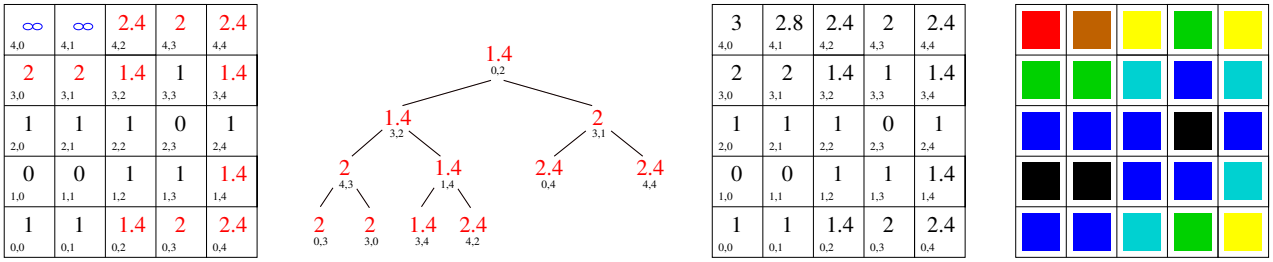


propagate fixed value to neighbors



adjust existing values & add new values to the priority queue

So we need to propagate the data from pixel (2,1) to its 8 neighbors. Two of the neighbors have known (black) distance values and can be ignored since their values won't change with this new information. Two of the neighbors had previously unknown distance estimates (marked blue) and are switched to be red pixels and added to the queue (shown with red boxes). Finally, four of the neighbors already have estimated distance values and if their values change through the propagation of distance data (the estimated distance value may only decrease), their position in the heap may also need to be adjusted (by calling `percolate_up`). To do so, we will need to quickly locate the position of that pixel within the heap. We certainly don't want to do a linear scan of the data in the heap to find that pixel. One solution is to use a map. In the priority queue representation, in addition to having a vector that stores the heap elements, we'll also use a map that corresponds each red pixel coordinate to its current index in the heap.



after fixing all pixels <= 1 | priority queue after fixing all pixels <= 1 | final distance field | output image

The two leftmost images above show an intermediate state of the algorithm for this example: after all pixels with distance value <= 1 have been marked fixed (black), propagated their distance data to their immediate neighbors, and been removed from the priority queue. The third diagram shows the final distance field values after all pixels have been fixed and the queue is empty and the fourth image shows the output image, in which each distance value is translated into an appropriate color in the rainbow scale. Note: This advancing front method produces an *approximation* of the true distance field. Because exact pairwise distances are only calculated for the 8 neighbors of a particular pixel, the calculated distance values of points far from the input shape may only be an upper bound of the true value.

We have provided starter code for the implementation of the Fast Marching Method using a priority queue for calculating the distance field function. Your task is to complete the implementation, test your code to be sure it approximately matches the naive algorithm, determine the order notation, and collect and analyze runtime data for a variety of input images.

For extra credit you may implement alternate visualization strategies for the distance field data (include sample images in your submission). Another option for extra credit is to use a hash table instead of a map to store the pixel to index correspondence within the priority queue. Collect and analyze data comparing the algorithm using a map to the algorithm using a hash table.

Viewing `.ppm` Images

The input and output files are stored in the simple uncompressed standard "Portable Pixel Map" format (with "magic number" P6), `.ppm`. Many standard image viewing programs (Photoshop, GIMP, and `xv`) will load and display these images. On UNIX/Linux/MacOSX/Cygwin, ImageMagick can be used to convert between image formats, such as the popular "Portable Network Graphics" format, `.png`. For example:

```
convert.exe tmp.ppm tmp.png          or          convert tmp.ppm tmp.png
```

Submission

Use the provided template `README.txt` file for your algorithm analysis and any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages with anyone, please list their names in your `README.txt` file.**